Scalability of Sparse Direct Solvers

Robert Schreiber

The Research Institute of Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 311, Columbia, MD 244, (301)730-2656

Work reported herein was supported in part by the NAS Systems Division of NASA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035.

SCALABILITY OF SPARSE DIRECT SOLVERS *

ROBERT SCHREIBER†

Abstract. We shall say that a scalable algorithm achieves efficiency that is bounded away from zero as the number of processors and the problem size increase in such a way that the size of the data structures increases linearly with the number of processors. In this paper we show that the column-oriented approach to sparse Cholesky for distributed-memory machines is not scalable. By considering message volume, node contention, and bisection width, one may obtain lower bounds on the time required for communication in a distributed algorithm. Applying this technique to distributed, column-oriented, full Cholesky leads to the conclusion that N (the order of the matrix) must scale with P (the number of processors) so that storage grows like P^2. So the algorithm is not scalable. Identical conclusions have previously been obtained by consideration of communication and computation latency on the critical path in the algorithm; these results complement and reinforce that conclusion.

For the sparse case, we have experimental measurements that make the same point: for column-oriented distributed methods, the number of gridpoints (which is $O(N)$) must grow as P^2 in order to maintain parallel efficiency bounded above zero. Our sparse matrix results employ the "fan-in" distributed scheme, implemented on machines with either a grid or a fat-tree interconnect using a subtree-to-submachine mapping of the columns.

The alternative of distributing the rows and columns of the matrix to the rows and columns of a grid of processors is shown to be scalable for the dense case. Its scalability for the sparse case has been established previously. To date, however, none of these methods has achieved high efficiency on a highly parallel machine.

Finally, open problems and other approaches that may be more fruitful are discussed.

Keywords. massively parallel computer, sparse Cholesky factorization, distributed-memory, scalable algorithms.

AMS(MOS) subject classifications: 65F50, 65F25, 68R10.

1. Introduction.

An efficient, highly parallel, distributed-memory, direct solution algorithm for the sparse linear system $Ax = b$ remains undiscovered, despite some prolonged and extensive investigations by a number of researchers [2, 3, 4, 9, 10, 14, 15, 18, 19, 30]. The arrival of highly and massively parallel supercomputers makes this an opportune time to decide whether or not to continue the search, perhaps along different lines, or to give it up in favor of iterative methods.

Two lines of attack have been taken up to now. The MIMD, message-passing-machine community has tended to concentrate on methods that are column oriented [2, 3, 4, 9, 14, 18, 19, 30]. In these methods, columns of the matrix A and its Cholesky factor L are assigned to processors in some way – column j is held by processor $map(j)$ and map is determined as part of the method. Furthermore, the methods organize the computation as a collection of column-oriented tasks: sparse column scaling and sparse DAXPY. This class of methods has also been proposed and used for the dense problem on message-passing machines [1].

A second approach is to map the data in two dimensions. This approach is favored by Gilbert and Schreiber [10], Kratzer [15], and Venugopal and Naik [29]. Recently, Dongarra, Van de Geijn, and Walker [7] have shown the value of this approach for the dense problem

* Written May 1992.

† Research Institute for Advanced Computer Science, MS T045-1 NASA Ames Research Center, Moffett Field, CA 94035. This author's work was supported by the NAS Systems Division via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA).

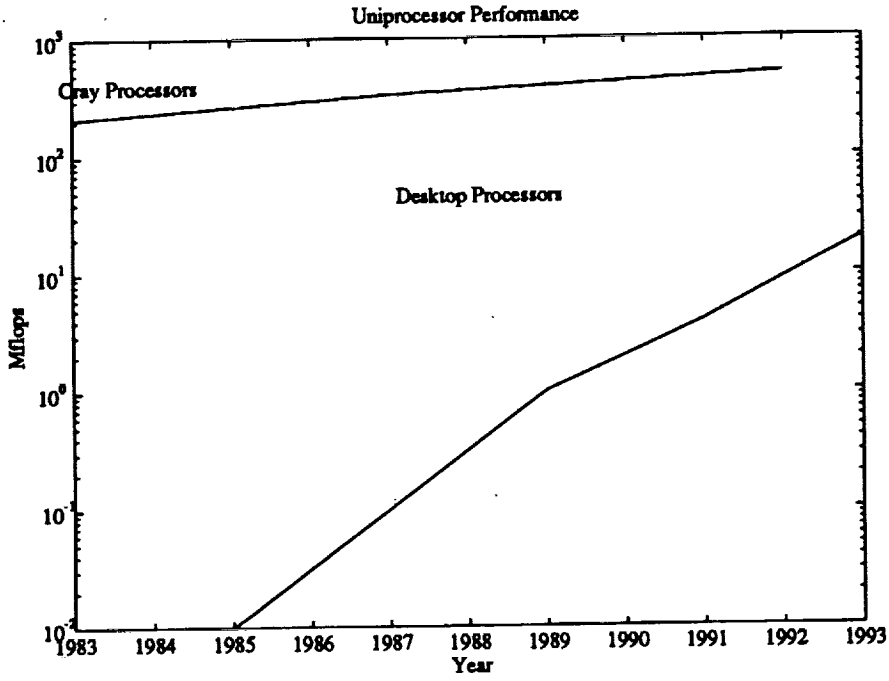

FIG. 1. *Microprocessor and supercomputer performance per CPU.*

on MIMD message passing machines; the author has also used it successfully for the dense problem on the Maspar MP-1, a massively parallel SIMD machine.

In this paper we investigate the scalability of these classes of methods for distributed sparse Cholesky factorization. By a *scalable* algorithm for this problem, we mean one that maintains efficiency bounded away from zero as the number P of processors grows and the problem size (in this case the number of gridpoints or the order of the matrix) grows roughly linearly in P. We concentrate on the model problem arising from the 5-point, finite difference stencil on an $N_g \times N_g$ grid. We will show that the column-oriented methods cannot work well when the number of gridpoints ($N = N_g^2$) grows like $O(P)$ or even $O(P \log P)$. We show that communication will make any column-oriented, distributed algorithm useless, *no matter what the mapping of columns to processors.* This is true because column-oriented distribution is very bad for *dense* problems of order N when N is not large compared with P.

Two improvements seem to be required.

1. A two-dimensional wrap mapping of the dense frontal matrices, at least for those corresponding to fronts near the top of the elimination tree.
2. A "fan-out" submatrix Cholesky algorithm with multicast instead of individual messages.

It is reasonable to ask why one should be concerned with machines having thousands of processors. Figure 1 should illustrate the reasons for believing that supercomputer architecture is now making an inevitable and probably permanent transition from the modestly

parallel to the highly parallel (257 – 4,096 processors) or massively parallel (4,097 – 65,536 processors). The following estimation of supercomputer architecture during the coming decade helps motivate the work presented here.

- 1 Gflop processor chips (with multiple processors);
- Physically distributed memory; while hardware may provide the illusion of shared memory, the latency for nonlocal access will be large and bandwidth to nonlocal memory will be a constraining resource.
- Communication speed between nodes will be on the order of 100 bits in parallel at 100 Mhz. (Since a sparse word is 12 bytes = 96 bits, roughly 100 Mwds/sec will be the achievable speed. Thus, the ratio of computation speed per processor to interprocessor communication speed will be in the 5 – 50 range. Patterson [23] gives comparable estimates.
- Interconnect may be a 2D or 3D grid or torus, or maybe a fat tree.

This paper builds on previous efforts at analysis of distributed matrix computations. The work of Leiserson [16] prefigures much later work of this type. Notable efforts for dense problems include those of Li and Coleman [17] for dense triangular systems, and Saad and Schultz [28]; Ostrouchov, *et al.* [22], and George, Liu, and Ng [8] have made some analyses for the sparse, column-mapped algorithms. An interesting analysis of the effect of a memory hierarchy on sparse Cholesky has been provided by Rothberg and Gupta [26]. These investigators, working with a nununiform-access shared-memory system, have recently come to conclusions similar to ours [27].

In Section 2 we introduce distributed implementations of Cholesky factorization; Section 3 develops some lower bounds on communication time; in Section 4 we compute these bounds for the dense case and use them to illustrate the problem with column mapping; Section 5 extends this work through an experiment for the sparse case; in Section 6 we consider the problems that are still unresolved.

2. Distributed sparse Cholesky. Cholesky factorization may be understood as the following program

cholesky(a, n)

for $k = 1$ **to** n **do**
  cdiv(k);
  for $j = k + 1$ **to** n **do**
    cmod(j, k);
  od
od

Procedure *cdiv*(k) computes the square root of the diagonal element A_{kk} and scales the k^{th} column of A by $1/\sqrt{A_{kk}}$ to produce the k^{th} factor column L_{*k}; procedure *cmod*(j, k) subtracts L_{jk} times the k^{th} column from the j^{th} column.

The execution order of this program is not the only one possible. The true dependences require only that *cmod*(j, k) must follow *cdiv*(k) and *cdiv*(k) must follow all the *cmod*(k, ℓ)

for $\ell < k$ and $L_{k\ell} \neq 0$. A second form of Cholesky is this:

cholesky(a, n)

```
for k = 1 to n do
    for ℓ = 1 to k − 1 do
        cmod(k, ℓ);
    od
    cdiv(k);
od
```

The first form is sometimes called "submatrix" Cholesky and sometimes called a "right-looking" method. The second form goes by the names "column" or "left-looking".

In the sparse case, sparsity is exploited within the vector *cdiv* and *cmod* operations. Furthermore, most *cmod* operations are omitted altogether because the multiplying scalar L_{jk} is zero.

The column-oriented distributed methods map columns to processors; column k is stored at processor $map[k]$. The operation $cdiv(k)$ is performed at $map[k]$. The operation $cmod(j, k)$ may be performed at $map[j]$, in which case the column of L_{*k} must be sent out from $map[k]$ after the $cdiv(k)$ is performed. This approach is known as a "fan-out" implementation. Alternatively, the $cmod(j, k)$ may be performed at $map[k]$, as follows. Consider the set of updates to column j. There is one update ($cmod(j, k)$) for each $k < j$ such that $L_{jk} \neq 0$. Processor π can compute the scaled column $L_{jk}L_{*k}$ for each such k for which $map[k] = \pi$. It then adds these scaled columns together to form an "aggregate update" vector $u[j, \pi]$ and sends this vector to processor $map[j]$. All communication is in the form of these aggregate updates. Whenever one arrives at a processor, it is subtracted from the updated column. This method is known as the "fan-in" distributed algorithm.

The node code for a fan-in method is shown in Figure 2. The data structure at processor π is the integer n-vector map, the columns of A and L mapped to π, The set $mycols = \{k \mid map[k] = \pi\}$, and the sets $row[j, \pi] = \{k \mid map[k] = \pi \text{ and } L_{jk} \neq 0\}, 1 \leq j \leq n$.

(As befits an MIMD code, the schedule of computation is not that presented in either of the sequential methods above. Instead computations occur at times that depend on the sequence of arriving data, and are not determined in advance.)

It is clear from this code that running time has an $O(N)$ term, because of the outer, sequential **for** loop. The running time of an efficient implementation should be at most $O(Ops/P)$. For a typical sparse problem, the number of floating-point operations is $Ops = O(N^{1.5})$, so running time should be at most $O(N^{1.5}/P)$; thus we cannot take $P > O(N^{.5})$. On the other hand, since the storage for the solution and the matrix grows like $O(N)$, and for the Cholesky factor like $O(N \log N)$, we would like to scale P as $O(N)$ or $O(N/\log N)$. Other scalability problems will be discussed in the following sections.

3. Methodology. Consider any distributed-memory computation. In order to assess the communication costs analytically, we have found it useful to employ certain abstract

```
fan − in(a, L, n, map)
integer n, map[];
real L[], a[];

    mycols = {j | map[j] = myname};
    for j = 1 to n do
        if ( row[j, myname] ≠ ∅ || j ∈ mycols ) then
            t = 0;
            for k ∈ row[j, myname] do
                t = t + a_jk(a_jk, ..., a_nk)^T ;
            od
            if ( j ∉ mycols ) then
                Send aggregate update column t to processor map[j]
            else
                L_*j = (a_jj, ..., , a_nj)^T − t;
                while not all aggregate updates have been received do
                    Receive an aggregate update column u[j, π] for column j;
                    L_*j = L_*j − u[j, π];
                od
                L_*j = L_*j / √(ℓ_jj);
            fi
        fi
    od
```

FIG. 2. *Fan-in distributed, column-oriented Cholesky.*

lower bounds. Our approach is neither new nor deep; it is a straightforward accounting for communication costs.

Our model assumes that machine topology is given. It assumes that memory consists of the memories local to processors. It assumes that the communication channels are the edges of a given undirected graph $G = (W, L)$, and that processor–memory units are situated at some, possibly all, of the vertices of the graph. The model includes hypercube and grid-structured message-passing machines, shared-memory machines having physically distributed memory (the Tera machine) as well as tree-structured machines like a CM-5.

Let $V \subseteq W$ be the set of all processors and L be the set of all communication links.

We assume identical links. Let β be the inverse bandwidth (slowness) of a link in seconds per word. (We ignore start-up costs in this model.)

We assume that processors are identical. Let ϕ be the inverse computation rate of a processor in seconds per floating-point operation. Let β_0 be the rate at which a processor can send or receive data, in seconds per word. We expect that β_0 and β will be roughly the same.

A distributed-memory computation consists of a set of processes that exchange information by sending and receiving messages. Let M be the set of all messages communicated. For $m \in M$, $|m|$ denotes the number of words in m. Each message m has a source processor $src(m)$ and a destination processor $dest(m)$, both elements of V.

For $m \in M$, let $d(m)$ denote the length of the shortest machine path from the source of the message m to its destination. We assume that each message takes a certain path of links from its source to its destination processor. Let $p(m) = (\ell_1, \ell_2, \ldots, \ell_{d(m)})$ be the path taken by message m. For any link $\ell \in L$, let the set of messages whose paths utilize ℓ, $\{m \in M \mid \ell \in p(m)\}$, be denoted $M(\ell)$.

The following are obviously lower bounds on the completion time of the computation. The first three bounds are computable from the set of message M, each of which is characterized by its size and its endpoints. The last depends on knowledge of the paths $p(M)$ taken by the messages.

1. (Flux per link)

$$\frac{\sum_{m \in M} |m| \cdot d(m)}{|L|} \cdot \beta.$$

2. (Bisection width) Given $V_0, V_1 \subseteq W$, V_0 and V_1 disjoint, define

$$sep(V_0, V_1) \equiv \min |\{L' \subseteq L \mid L' \text{ is an edge separator of } V_0 \text{ and } V_1\}|$$

and

$$flux(V_0, V_1) \equiv \sum_{\{m \in M \mid src(m) \in V_i, dest(m) \in V_{1-i}\}} |m|.$$

The bound is

$$\frac{flux(V_0, V_1)}{sep(V_0, V_1)} \cdot \beta.$$

3. (Arrivals/Departures (also known as node congestion))

$$\max_{v \in V} \sum_{dest(m) = v} |m|\beta_0;$$

$$\max_{v \in V} \sum_{src(m) = v} |m|\beta_0.$$

4. (Edge contention)

$$\max_{\ell \in L} \sum_{m \in M(\ell)} |m|\beta.$$

Of course, the actual communication time may be greater than any of the bounds. In particular, the communication resources (the wires in the machine) need to be scheduled. This can be done dynamically or, when the set of messages is known in advance, statically. With detailed knowledge of the schedule of use of the wires, better bounds can be obtained. For the purposes of analysis of algorithms and assignment of tasks to processors, however, we have found this more realistic approach to be unnecessarily cumbersome. We prefer to use the four bounds above, which depend only on the integrated (i.e. time-independent) information M and, in the case of the edge-contention bound, the paths $p(M)$.

4. Dense Cholesky. In this section we consider dense, distributed Cholesky factorization. Since, for the model problem, a constant and substantial fraction of the work in a sparse Cholesky factorization is spent doing a final dense Cholesky factorization of a matrix of order N_g, efficiency on this final dense problem is a *sine qua non* for a scalable sparse algorithm.

4.1. Mapping columns. Let us consider a right-looking, fan-out distributed Cholesky. Assume that the columns of a dense symmetric matrix of order N are mapped to processors cyclically: column j is stored in processor $map(j) \equiv j \bmod P$.

We first examine the parallelism and the critical path. Execution time must be at least $N^2\phi \max\left(\frac{3}{2}, \frac{N}{3P}\right)$, no matter the scheduling of the tasks. The second term comes from the operation count, $N^3/3$. The first is due to the longest path in the computation DAG, which has $N^2/2$ multiplies and multiply-adds. This is the path $cdiv(1), cmod(2,1), cdiv(2), cmod(3,2), \ldots$. By making column operations an atomic unit of computation, we have lengthened the critical path from $O(N)$ to $O(N^2)$ operations. Therefore, at most $O(N)$ processors can be used efficiently.

Next, consider communication costs on two-dimensional grid or toroidal machines. Suppose that P is a perfect square and that the machine is a $\sqrt{P} \times \sqrt{P}$ grid. (This assumption is not necessary for our conclusions, but it simplifies things.)

Consider a mapping of the computation in which the operation $cmod(j,k)$ is performed by processor $map(j)$ (a fan-out method). After performing the operation $cdiv(k)$, processor $map(k)$ must send column k to all processors $\{map(j) \mid j > k\}$.

	Grid	Torus
2D	$(2/3)\sqrt{P}$	$(1/2)\sqrt{P}$
3D	$P^{1/3}$	$(3/4)P^{1/3}$

TABLE 1
Average interprocessor distances.

Two possibilities present themselves. These sends may be done separately and sequentially by processor $map(k)$, with separate messages each taking its own path to the several destinations; or they may be sent through a spanning tree of the processor graph, whose root is processor $map(k)$ and whose nodes include the destination processors.

In order to compute flux per link, we need to know the average path length traversed by the messages. Let us assume that N is greater than P. (Otherwise we clearly have idle processors.) We first assume that the average message distance is just the average distance between two randomly chosen processors in the machine. For a mesh in 2D this is $(2/3)\sqrt{P}$; for a 2D torus it is $(1/2)\sqrt{P}$. In 3D the square roots become cube roots and the constants change to 1 for grids and (3/4) for tori (Table 1). Even if we are clever about assigning data to processors, and we place the early, large columns in the middle of the grid, we can at best reduce the average distances by a modest constant factor. So we will stick to the estimate based on random positions for source and destination.

Let us fix our attention on 2D grids. If separate messages are sent, the total flux is $(1/3)N^2P^{3/2}$. There are a total of $|L| = 2P$ links; the total machine bandwidth is roughly $2P/\beta$ and the flux-per-link bound is $(1/6)N^2\sqrt{P}\beta$ seconds.

With spanning tree multicast, the "average distance" computation changes. Most of the sends will use a tree of total length P reaching all the processors. Every matrix element will therefore travel over P links, so the total information flux is $(1/2)N^2P$ and the flux per link bound is $(1/4)N^2\beta$ seconds.

With multicast, only $O(N^2/P)$ words leave any processor. If $N \gg P$, processors see almost the whole $(1/2)N^2$ words of the matrix as arriving factor columns. The bandwidth per processor is β_0, so the arrivals bound is $(1/2)N^2\beta_0$ seconds. If $N \approx P$ the bound drops to half that, $(1/4)N^2\beta_0$ seconds.

Consider a bisection of the machine through its vertical midline. Since most sends must arrive at all processors, we may approximate the flux across the line by assuming that every factor column crosses. With individual messages, it crosses $(1/2)P$ times, for a total flux of $(1/4)N^2P$ words. With spanning tree multicast, the shape of the tree plays a role. The number of crossings is at least one. This obsdervation leads to a weak bound. Insstead, we will use a more realistic estimate that is not in fact a bound. A realistic assumption is that on average the tree intersects the cut in $(1/2)\sqrt{P}$ edges, since the tree uses half of all edges and there are $\sqrt{P}$ of them in the cut. Thus the flux is $(1/4)N^2\sqrt{P}$ words. The resulting lower bounds are $(1/4)N^2\sqrt{P}\beta$ seconds with separate messages and $(1/4)N^2$ seconds with tree multicast.

We summarize these bounds for 2D grids in Table 2.

Type of Bound	Lower bound	Communication Scheme
Arrivals	$\frac{N^2}{8}\beta_0$	
Flux per link	$\frac{N^2}{4}\beta$	tree multicast
Flux per link	$\frac{N^2}{6}\sqrt{P}\beta$	separate messages
Bisection width	$\frac{N^2}{4}\beta$	tree multicast
Bisection width	$\frac{N^2}{4}\sqrt{P}\beta$	separate messages

TABLE 2
Communication Costs for Column-Mapped Full Cholesky.

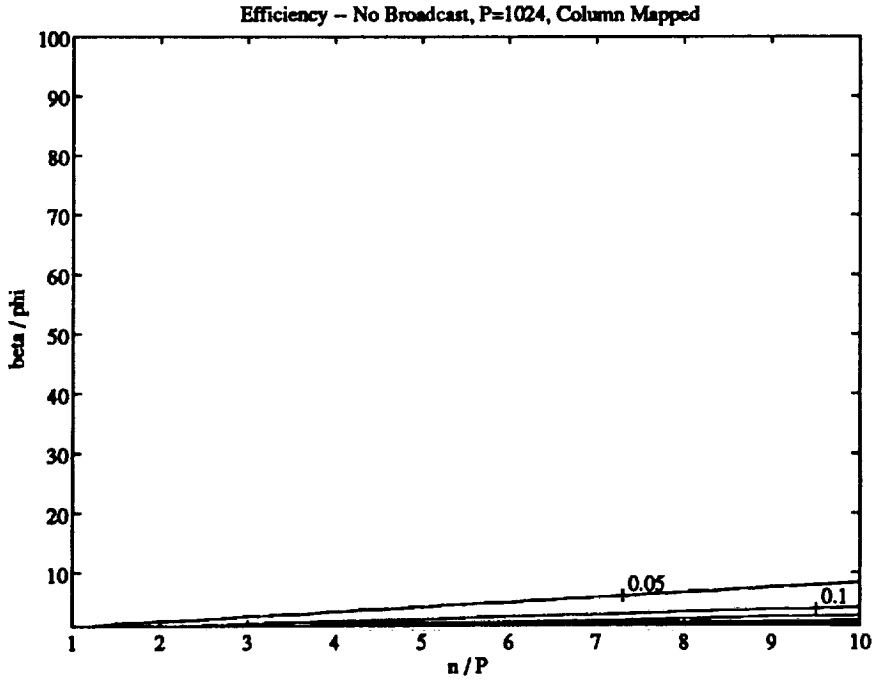

FIG. 3. *Iso-efficiency lines for dense Cholesky with column cyclic mapping; separate messages.*

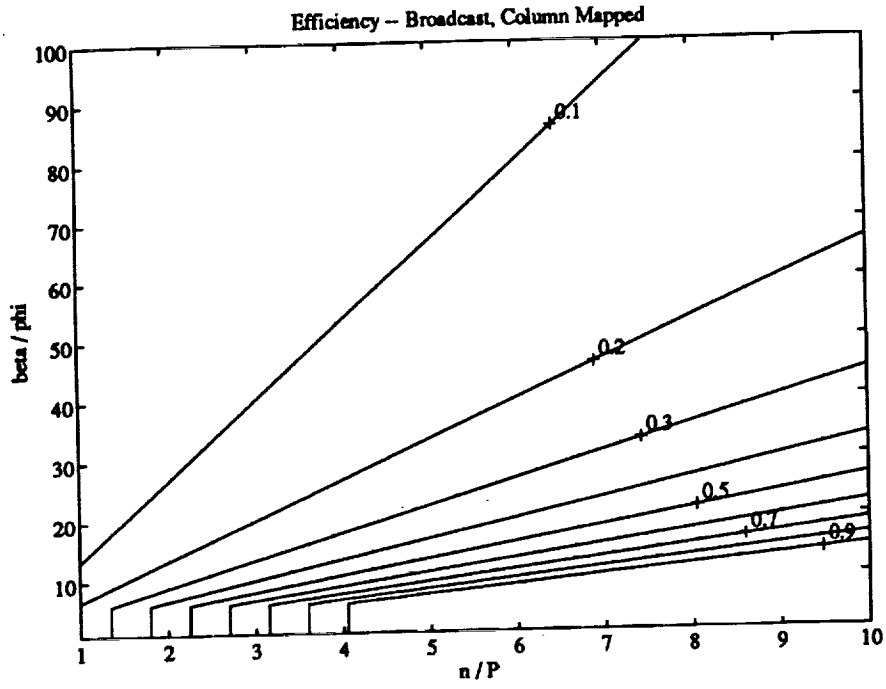

FIG. 4. *Iso-efficiency lines for dense Cholesky with column cyclic mapping, $P = 1,024$; tree multicast.*

From the critical path, average work per processor, and the bisection width bounds, we have that the completion time is roughly $\max(\frac{N^3\phi}{3P}, \frac{3N^2\phi}{2}, \frac{N^2\beta}{4})$ with tree multicast and $\max(\frac{N^3\phi}{3P}, \frac{3N^2\phi}{2}, \frac{N^2\sqrt{P}\beta}{4})$ with separate messages. Contours of efficiency (in the case $P = 1,024$) are shown in Figures 3 and 4.

We can immediately conclude that without spanning tree multicast, this is a nonscalable distributed algorithm. We suffer a loss of efficiency as P is increased, with speedup limited to $O(\sqrt{N})$. Even with spanning tree multicast, we may not take $P > \frac{N\phi}{\beta}$ and still achieve high efficiency. For example, with $\beta = 10\phi$ and $P = 1,000$, we require $N > 12,000$ (72,000 matrix elements per processor) in order to achieve 50% efficiency. This is excessive for full problems and will prove to be excessive in the sparse case, too.

4.2. Mapping blocks. Dongarra, Van de Geijn, and Walker have already shown that on the Intel Touchstone Delta machine ($P = 528$), mapping blocks is better than mapping columns. In such a mapping, we view the machine as an $P_r \times P_c$ grid and we map elements A_{ij} and L_{ij} to processor $(mapr(i), mapc(j))$. We assume a cyclic mappings here: $mapr(i) \equiv i \bmod P_r$ and similarly for $mapc$. In a right-looking method, two portions of column k are needed to update the block $A_{rows,cols}$: $L_{rows,k}$ and $L_{cols,k}$ ($rows$ and $cols$ are integer vectors here). Again, we may send the data in the form of individual messages from the P_r processors holding the data to those processors that need it, or we may use multicast.

The analysis of the preceding section may now be done for this mapping. Now the compute time must be at least $N^2\phi \max\left(\frac{3}{2P_r}, \frac{N}{3P}\right)$; the longest path in the task graph has

Type of Bound	Lower bound	Comment
Arrivals	$\frac{N^2\beta}{4}\left(\frac{1}{P_r}+\frac{1}{P_c}\right)$	
Edge contention	$N^2\beta\left(\frac{1}{P_r}+\frac{1}{P_c}\right)$	tree multicast
Edge contention	$\frac{N^2\beta}{2}\left(\frac{P_c}{P_r}+\frac{P_r}{P_c}\right)$	separate messages

TABLE 3
Communication Costs for Torus-Mapped Full Cholesky.

$N^2/2P_r$ multiplies and multiply-adds. For the multicast approach, the spanning trees are linear connections of the processors in machine rows and columns. With this information about the paths $p(m)$ taken by messages, we may compute the use of the most heavily loaded edge. This bound dominates the flux per link and bisection width. Results are summarized in Table 3. With P_r and P_c both $O(\sqrt{P})$, the communication time drops like $O(P^{-1/2})$. With this mapping and with efficient multicast, the algorithm is scalable even when $\beta \gg \phi$. Note that $P = O(N^2)$ so that storage per processor is $O(1)$. (In fact, this scalable algorithm for distributed Cholesky is due to O'Leary and Stewart in 1985 [21].)

Contours of efficiency for $P = 1,024$ and $P_r = P_c = 32$ are shown in Figures 5 and 6.

5. Distributed sparse Cholesky and the model problem. The interesting questions are about the sparse and not the full case. The best way to extend the results of the last section to the sparse case would be to do just that. But an analytical approach, even for the model problem, proved to be dauntingly complicated.

Instead, an experimental simulation was done and statistics collected. The experiment simulates (on a Sun workstation) the fan-in, distributed, column-oriented sparse Cholesky described above. The software used was Matlab, version 4.0, which has sparse matrix operations and storage [11].

First, the sparse Laplacian for an $N_g \times N_g$ grid was generated and factored, in order to obtain the structure of the factor L. The elimination tree was then computed. Finally, columns were "assigned" to processors by the subtree-to-submesh mapping as follows:

1. the top-level separator was mapped cyclically to the whole machine;
2. the left subtree was mapped recursively to the left half-machine;
3. the right subtree was mapped recursively to the right half-machine;

The number of the processor assigned to column k is computed and stored in $map(k)$. The matrix L and the vector map are all that is needed by a simulation that collects the time-integrated statistics

- a vector of operation counts per processor;
- a vector of counts of arriving words per processor;
- the total flux of data in word-links;
- the flux of data (in words) crossing the horizontal and vertical midlines of the machine.

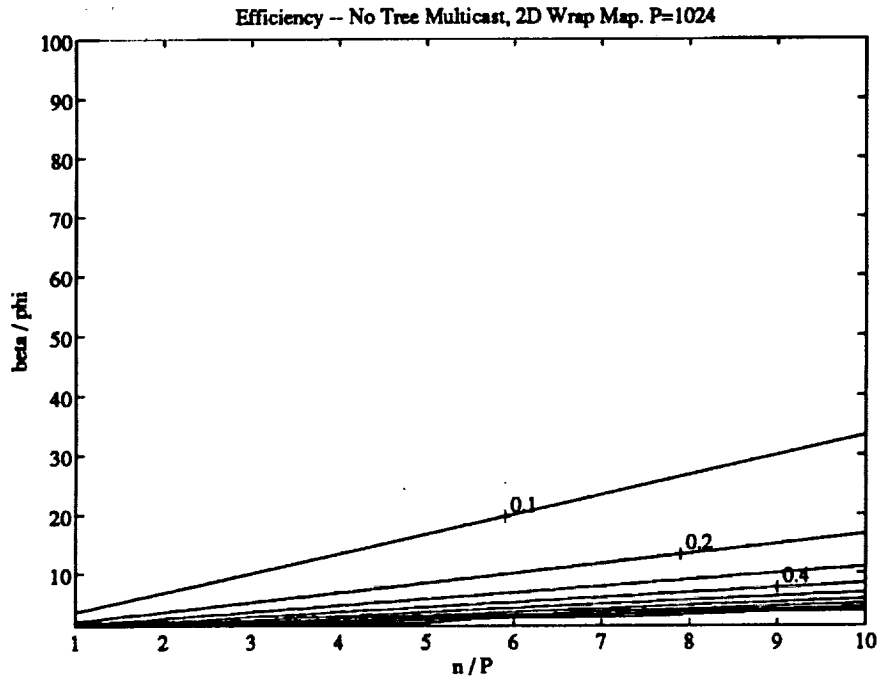

FIG. 5. *Iso-efficiency lines for dense Cholesky with 2D cyclic mapping; separate messages.*

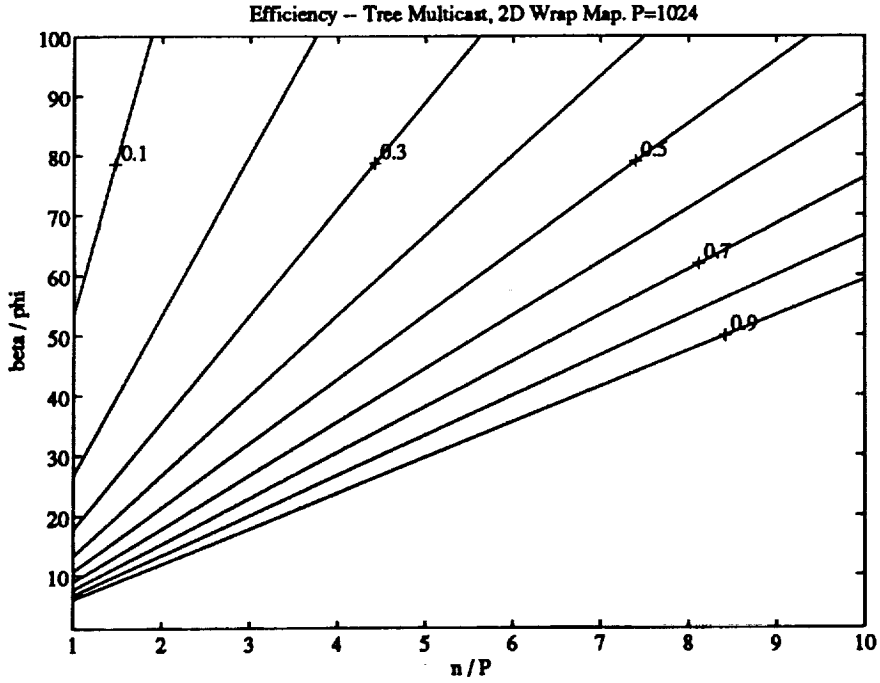

FIG. 6. *Iso-efficiency lines for dense Cholesky with 2D cyclic mapping; tree multicast.*

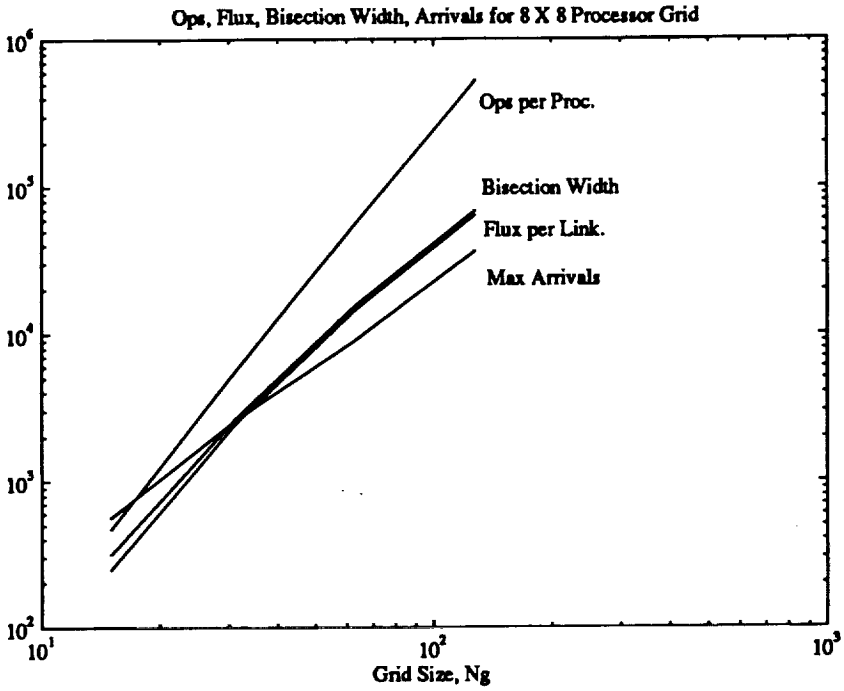

FIG. 7. *Four lower bounds;* $Pr = Pc = 8$.

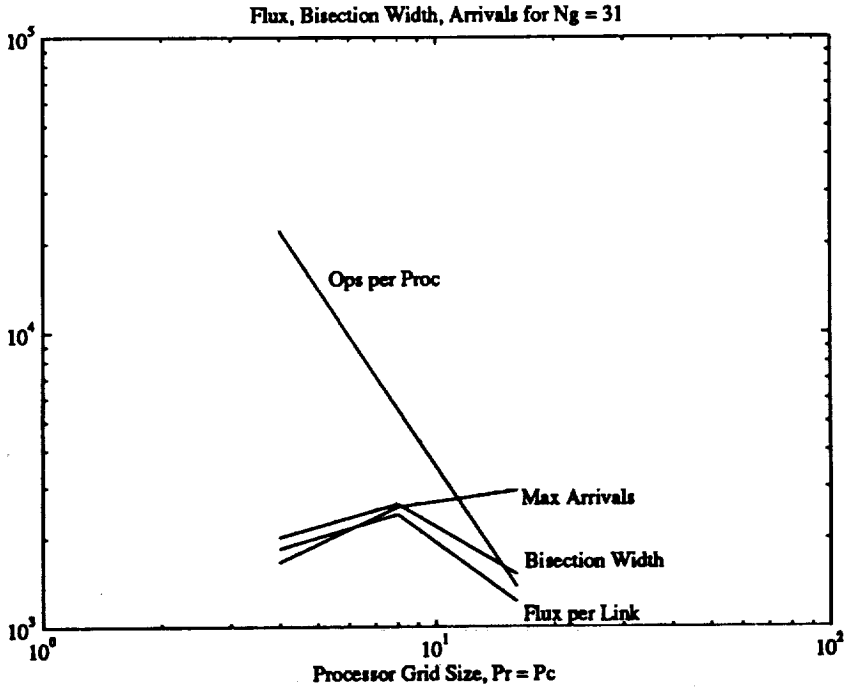

FIG. 8. *Four lower bounds;* $N_g = 31$.

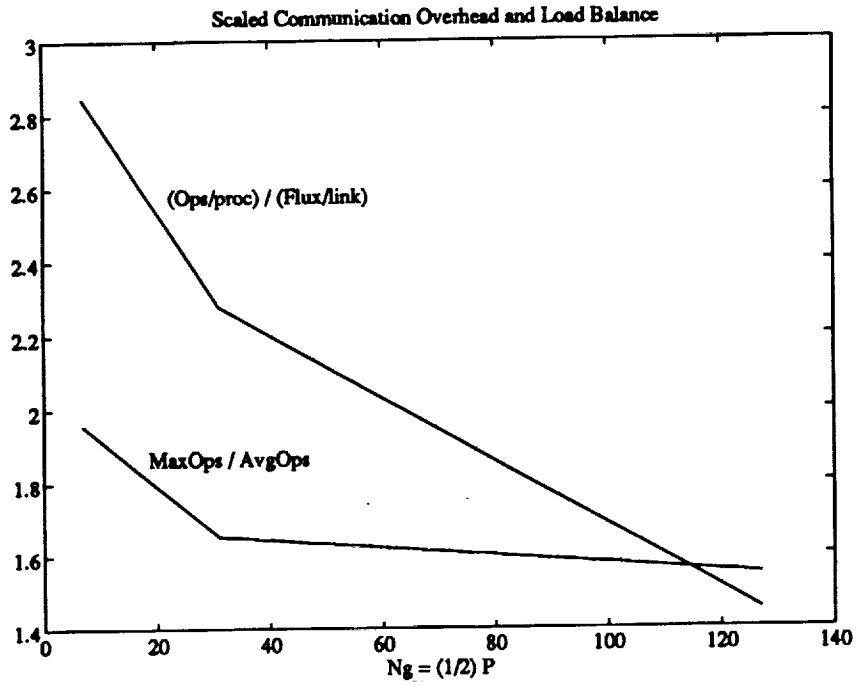

FIG. 9. *Scaled communication and load balance with* $N_g = (1/2)P$.

Figure 7 shows the computational load on the processors (Ops per Proc.), the bisection width bound, the maximum number of words arriving at any processor and the average flux of words per machine link as a function of the grid size N_g with $P_r = P_c = 8$; there are three data points on each curve, for grids of size $N_g = 15$, 31, and 63. The slope of the Ops per Processor curve is greater than that of the communication curves, as expected, and when $N_g \gg P$ efficiency will be good.

Figure 8 shows the behavior of these four metrics as P increases and N_g is fixed at 31. Now, the operations per processor curve drops as $1/P$, the communication curves do not, and efficiency is very poor when P is not much smaller than N_g.

The results for the dense case lead us to suspect that efficiency will be roughly constant if the ratio N_g/P is fixed. Figures 9 and 10 show two measures of efficiency over a range of values of N_g and P, with the ratio fixed at one half and at two. These curves are nearly flat, which confirms the main result of this work: one must scale the number of gridpoints as the *square* of the number of processors in order to have efficiency bounded above zero as P is increased. Thus, the method is not scalable by our earlier definition.

Recently, Thinking Machines Corporation has introduced a highly parallel machine with a "fat-tree" interconnect scheme. A fat tree is a binary tree of nodes. Leaves are processors and internal nodes are switches. The link bandwidth increases geometrically with increasing distance from the leaves.

We simulated column-mapped sparse Cholesky for a fat tree with bandwidth that doubles at each tree level. Columns were mapped in a subtree-to-subtree manner:

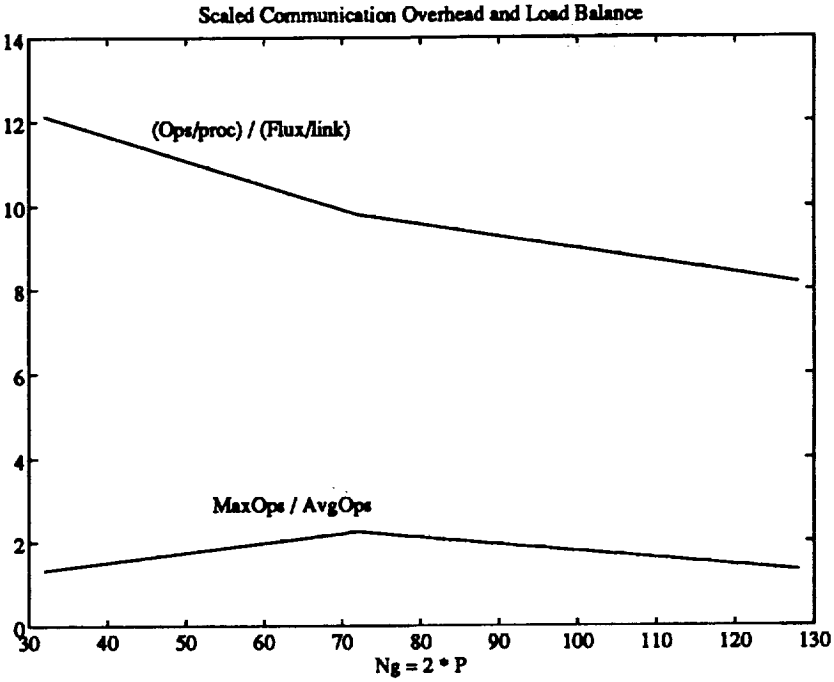

FIG. 10. *Scaled communication and load balance with* $N_g = 2P$.

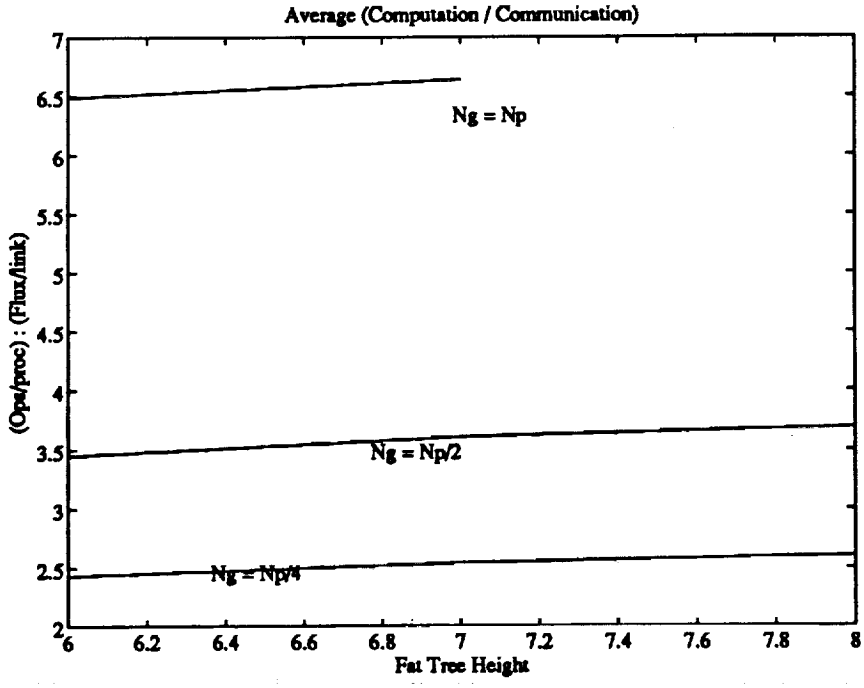

FIG. 11. *Scaled communication and load balance for fat trees, with* $N_g \propto P$.

1. the top-level separator was mapped cyclically to the whole machine;
2. the left subtree was mapped recursively to the left half-machine;
3. the right subtree was mapped recursively to the right half-machine;

The same set of statistics were collected; they are shown in Figure 11. Clearly, our conclusions hold for fat trees as well as meshes. Perhaps this is surprising, since average interprocessor distance is now $O(\log P)$ and the bisection bandwidth of the machine is $O(P)$ instead of $O(\sqrt{P})$. This is additional evidence that column-mapped methods are not scalable for highly parallel machines.

6. **Further work.** This work should be extended in several ways.
   - Experimental performance data should be taken from actual distributed dense and sparse Cholesky and compared with our predictions.
   - Variants that map the sparse matrix data in some form of two-dimensional cyclic map, as has been suggested by Gilbert and the author, Kratzer, and by Venugopal and Naik, should also be scrutinized experimentally.
   - The whole Cholesky factorization can be viewed as a DAG whose nodes are arithmetic operations and edges are values. (An n-input SUM operator should be used so as not to predefine the order of updates to an element.) Let us call this the computation DAG. The ultimate problem is to assign all the nodes to processors in such a way that the completion time is minimized.
     The computation DAG is quite large. Methods that work with an uncompressed representation of this DAG suffer from excessive storage costs. (This idea is quite like the very old one of generating straight-line code for sparse Cholesky in which the size of the program is proportional to the number of flops, and hence is larger than the matrix and its factor.)
     Of course, Cholesky DAGs have underlying regularity that allows for compressed representations. One such representation is the structure of L. Others, smaller still, have been derived from the supernodal structure of L and are usually only as large as a constant multiple of the size of A.
     All approaches to the problem to date have employed an assignment of computation to processors that is derived from the structure of L rather than from the computation DAG. None has succeeded. It is not known, however, if this failure is due to a poor choice of assignment, or alternatively if *any* assignment based only on the structure of L must in some way fail, or indeed whether there is *any* assignment for sparse Cholesky computation DAGs that will succeed. These issues require some investigation.
   - In these proceedings, Ashcraft proposes a new class of column-oriented methods in which the assignment of work to processors differs from the assignment used in the algorithms we have investigated. His approach may make for a substantial reduction in the fulx per link and bisection width requirements of the method, and so it should be investigated further. We note, however, that it will not reduce the length of the critical path, since it is based on the same task graph as all column-oriented methods.
   - It appears that the scalable implementation of iterative methods is much easier than it is for sparse Cholesky. Indeed, even naive distributed implementation of attractive

iterative methods is quite efficient. For example, with a regular grid, simple mappings of gridpoints to processors allows fast calculation of matrix-vector products. Total flux is kept to a small fraction of the operation count by mapping compact subgrids to processors, so that most edges of the grid connect gridpoints that reside on the same processor. Recent work of Hammond [12], Pommerell, Annaratone, and Fichtner [24], and Pothen, Simon, and Wang [25] makes it clear that this can be done, at some noticeable but supportable preprocessing cost, even for irregular grids. When Krylov subspace methods are used, dot products may be annoying; but all that we require to make them tolerable is, at worst, that the number of gridpoints grow like $P \log P$, not P^2. Useful, fully parallel preconditioners have also been developed. Finally, domain decomposition methods (which can be viewed as the class of preconditioned Krylov subspace methods designed to take advantage of spatial locality) are even more suitable in the distributed-memory environment. A good example of the power of parallel domain decomposition methods has recently been provided by Bjørstad and Skogen [6], who found that $P = 16,384$ was no impediment to the efficient solution of finite difference equations with N_g equal to only 640.

- We conclude by admitting that it is not yet clear whether sparse direct solvers can be made competitive at all for highly ($P > 256$) and massively ($P > 4096$) parallel machines.

REFERENCES

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau and R. van de Geijn, *LAPACK for distributed memory architectures: progress report,* In Parallel Processing for Scientific Computing, SIAM, 1992.

[2] C. Ashcraft, S. C. Eisenstat, and J. W. H. Liu, *A fan-in algorithm for distributed sparse numerical factorization,* SIAM J. Scient. Stat. Comput. 11 (1990), pp. 593–599.

[3] C. Ashcraft, S. C. Eisenstat, J. W. H. Liu, and A. H. Sherman, *A comparison of three column-based distributed sparse factorization schemes,* Research Report YALEU/DCS/RR-810, Comp. Sci. Dept., Yale Univ., 1990.

[4] C. Ashcraft, S. C. Eisenstat, J. W. H. Liu, B.W. Peyton, and A. H. Sherman, *A compute-ahead fan-in scheme for parallel sparse matrix factorization,* In D. Pelletier, editor, Proceedings, Supercomputing Symposium '90, pp. 351–361. École Polytechnique de Montréal, 1990.

[5] C. Ashcraft, *The fan-both family of column-based distributed Cholesky factorization algorithms,* These proceedings.

[6] P. Bjørstad and M. D. Skogen, *Domain decomposition algorithms of Schwarz type,* designed for massively parallel computers. Proceedings of the Fifth International Symposium on Domain Decomposition. SIAM, 1992.

[7] J. Dongarra, R van de Geijn, and D. Walker, *A look at scalable dense linear algebra libraries,* Proceedings, Scalable High Performance Computer Conference, Williamsburg, VA, 1992.

[8] A. George, J. W. H. Liu, and E. Ng, *Communication results for parallel sparse Cholesky factorization on a hypercube,* Parallel Comput. 10 (1989), pp. 287–298.

[9] A. George, M. T. Heath, J. W. H. Liu, and E. Ng, *Solution of sparse positive definite systems on a hypercube,* J. Comput. Appl. Math. 27 (1989), pp. 129–156.

[10] J. R. Gilbert and R. Schreiber, *Highly parallel sparse Cholesky factorization,* SIAM J. Scient. Stat. Comput. , to appear.

[11] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: design and implementation,* SIAM J. Matrix Anal. Appl. 13 (1992), pp. 333–356.

[12] S. W. HAMMOND, *Mapping Unstructured Grid Computations to Massively Parallel Computers,* PhD thesis, Dept. of Comp. Sci., Rensselaer Polytechnic Institute, 1992.

[13] S. W. HAMMOND AND R. SCHREIBER, *Mapping unstructured grid problems to the Connection Machine,* In Piyush Mehrotra, J. Saltz, and R. Voigt, editors, Unstructured Scientific Computation on Multiprocessors, pp. 11–30. MIT Press, 1992.

[14] M. T. HEATH, E. NG, AND B. W. PEYTON, *Parallel algorithms for sparse linear systems,* SIAM Review 33 (1991), pp. 420–460.

[15] S. G. KRATZER, *Massively parallel sparse matrix computations,* In P. Mehrotra, J. Saltz, and R. Voigt, editors, Unstructured Scientific Computation on Multiprocessors, pp. 178–186. MIT Press, 1992. A more complete version will appear in J. Supercomputing.

[16] C. E. LEISERSON, *Fat-trees: universal networks for hardware-efficient supercomputing,* IEEE Trans. Comput. C-34 (1985), pp. 892–901.

[17] GUANGYE LI AND THOMAS F. COLEMAN, *A parallel triangular solver for a distributed memory multiprocessor,* SIAM J. Scient. Stat. Comput. 9 (1988), pp. 485–502.

[18] M. MU AND J. R. RICE, *Performance of PDE sparse solvers on hypercubes,* In P. Mehrotra, J. Saltz, and R. Voigt, editors, Unstructured Scientific Computation on Multiprocessors, pp. 345–370. MIT Press, 1992.

[19] M. MU AND J. R. RICE, *A grid based subtree-subcube assignment strategy for solving PDEs on hypercubes,* SIAM J. Scient. Stat. Comput. , 13 (1992), pp. 826–839.

[20] A. T. OGIELSKI AND W. AIELLO, *Sparse matrix algebra on parallel processor arrays,* These proceedings.

[21] D. P. O'LEARY AND G. W. STEWART, *Data-flow algorithms for parallel matrix computations,* Comm. ACM, 28 (1985), pp. 840–853.

[22] L.S. OSTROUCHOV, M.T. HEATH, AND C.H. ROMINE, *Modelling speedup in parallel sparse matrix factorization,* Tech Report ORNL/TM-11786, Mathematical Sciences Section, Oak Ridge National Lab., December, 1990.

[23] D. PATTERSON, *Massively parallel computer architecture: observations and ideas on a new theoretical model,* Comp. Sci. Dept., Univ. of California at Berkeley, 1992.

[24] C. POMMERELL, M. ANNARATONE, AND W. FICHTNER, *A set of new mapping and coloring heuristics for distributed-memory parallel processors,* SIAM J. Scient. Stat. Comput. 13 (1992), pp. 194–226.

[25] A. POTHEN, H. D. SIMON, AND L. WANG, *Spectral nested dissection,* Report CS-92-01, Comp. Sci. Dept., Penn State Univ. Submitted to J. Parallel and Distrib. Comput.

[26] E. ROTHBERG AND A. GUPTA, *The performance impact of data reuse in parallel dense Cholesky factorization,* Stanford Comp. Sci. Dept. Report STAN-CS-92-1401.

[27] E. ROTHBERG AND A. GUPTA, *An efficient block-oriented approach to parallel sparse Cholesky factorization,* Stanford Comp. Sci. Dept. Tech. Report, 1992.

[28] Y. SAAD AND M.H. SCHULTZ, *Data communication in parallel architectures,* Parallel Comput. 11 (1989), pp. 131–150.

[29] S. VENUGOPAL AND V. K. NAIK, *Effects of partitioning and scheduling sparse matrix factorization on communication and load balance,* Proceedings, Supercomputing 91, pp. 866–875. IEEE Computer Society Press, 1991.

[30] E. ZMIJEWSKI, *Limiting communication in parallel sparse Cholesky factorization,* Tech. Report TRCS89-18, Dept. of Comp. Sci., Univ. of California, Santa Barbara, CA, 1989.